

1.8 Documentation with Comments

Self-Study Guide · About 50 minutes

By the end you will be able to

- Explain who comments are written for and why they never change what a program does.
- Read and write single-line, block, and Javadoc comments, and predict the output of code that contains them.
- Explain how Javadoc comments become a library's documentation, and read the @param and @return lines in them.
- State the precondition of a method or code segment, and explain who is responsible for making it true.
- State the postcondition of a method or code segment, and tell it apart from a precondition.

What to remember from earlier

- A run-time error such as dividing by zero stops a running program early. ([Topic 1.1: Introduction to Algorithms, Programming, and Compilers, 1.1.8 Run-Time Errors and Exceptions](#))
- With two int values, / keeps the whole-number part and % gives the remainder. ([Topic 1.3: Expressions and Output](#))
- (int) (x + 0.5) rounds a nonnegative double to the nearest whole number. ([Topic 1.5: Casting and Range of Variables, 1.5.4 Rounding to the Nearest Integer](#))
- x += 3 means x = x + 3, and *= and -= work the same way. ([Topic 1.6: Compound Assignment Operators, 1.6.1 Adding to a Variable in One Step](#))
- A library is a collection of classes, and a class has attributes and behaviors. ([Topic 1.7: Application Program Interface \(API\) and Libraries](#))

1.8.1 Comments Are Notes for People

Open a program you wrote two months ago and you may not remember why a line says `quizScore += bonus` or what the `2` in `int bonus = 2;` stands for. Now imagine reading a program someone else wrote. Java code says exactly *what* the computer does, but it rarely says *why*. The reasons live in the programmer's head, and nobody else can see them.

The fix is to write the reasons down, right next to the code, in plain English. Java gives you a way to do this so the compiler does not try to read your English as Java. In Topic 1.1 you saw that the compiler checks every line and translates it before the program runs. Text marked as a note is skipped by the compiler entirely: it is never checked, never translated, and never run.

A **comment** is text in a program that is written for people to read. Comments help the original programmer (you, later) and other programmers understand what the code does and why. The compiler ignores comments, so they are not part of the running program at all.

KEY IDEA

Comments are notes for people. The compiler ignores them, so adding, removing, or changing a comment never changes what a program does.

The quickest kind of comment starts with two slashes, `//`: everything from `//` to the end of that line is a comment. (1.8.2 looks at this kind closely.)

WORKED EXAMPLE The same program with and without comments

Here is a short program with no comments. Read it and ask yourself: what is the 18? What is the 2? Is a score above 20 a mistake?

ScoreReport.java

```
1 public class ScoreReport
2 {
3     public static void main(String[] args)
4     {
5         int quizScore = 18;
6         int bonus = 2;
7         quizScore += bonus;
8         System.out.print("Final quiz score: ");
9         System.out.println(quizScore);
10    }
11 }
```

Final quiz score: 20

Now the same statements with comments added on lines 5, 7, and 9:

ScoreReportCommented.java

```
1 public class ScoreReportCommented
2 {
3     public static void main(String[] args)
4     {
5         // Points earned on the quiz, out of 20.
6         int quizScore = 18;
7         // Extra credit for showing all work.
8         int bonus = 2;
9         // The bonus may push the score above 20. That is allowed.
10        quizScore += bonus;
11        System.out.print("Final quiz score: ");
12        System.out.println(quizScore);
13    }
14 }
```

Final quiz score: 20

Step by step, here is what the compiler and the computer do with the second version:

1. The compiler reaches line 5, sees `//`, and skips the rest of the line. The same happens on lines 7 and 9.
2. What is left to translate is exactly the five statements from the first program.

- When the program runs, it stores 18 in `quizScore`, stores 2 in `bonus`, adds the bonus (`quizScore` becomes 20), and prints the label and the score.

The output is identical. The only difference is for the human reader: the comments answer all three questions (18 is the quiz score out of 20, 2 is extra credit, and going above 20 is allowed).

Now change one thing: put a Java statement inside a comment. Programmers do this on purpose to switch a line off without deleting it, which is called **commenting out** the line.

```
1 int tickets = 3;
2 // System.out.println("Checking tickets...");
3 tickets *= 2;
4 System.out.println(tickets);
```

6

Line 2 is a perfectly good `println` statement, but it sits after `//`, so the compiler treats it as a note and the program never prints `Checking tickets...`. Only line 4 prints, showing `tickets` after it was doubled. The rule depends on *where* the text is, not on *what* the text looks like: anything inside a comment is ignored, even if it is valid Java.

WATCH OUT

Trusting a comment over the code. Because the compiler never reads comments, it cannot tell you when a comment is wrong. Read this carefully:

```
1 int score = 80;
2 // Add 10 bonus points.
3 score += 5;
4 System.out.println(score);
```

85

A student who reads only the comment predicts 90. But the comment is out of date: someone changed the bonus to 5 and forgot to update the note. The program does what the code says, so it prints 85. When a comment and the code disagree, the code wins. Whenever you change code, update its comment in the same moment, or you leave a trap for the next reader.

PAUSE AND TRY

Question 1 Multiple choice

A programmer adds ten comments to a program that already compiles and prints 42. Nothing else changes. What happens?

- A. The program still prints 42, because the compiler ignores comments.
- B. The program prints the comments along with 42.
- C. The program runs more slowly, because the computer has to read each comment.
- D. The program no longer compiles, because comments are not Java statements.

Question 2 Tracing

What does this code print?

```
1 int laps = 4;
2 // laps += 3;
3 laps *= 2;
4 // System.out.println(laps);
5 laps -= 1;
6 System.out.println(laps);
```

1.8.2 Single-Line Comments

Most notes a programmer needs are short: what a variable stands for, the units of a number, why a step is there. For those, a whole paragraph is too much. You want a note that fits on one line, sometimes on the same line as the code it explains.

A **single-line comment** starts with `//` and ends at the end of that line. It can take up a whole line by itself, or it can come after code on the same line, which is called an end-of-line comment.

```
1 // A whole-line comment: the entire line is a note.
2 int count = 3; // An end-of-line comment: the code runs, the note does not.
```

Take the second line apart:

- `int count = 3;` comes before `//`, so it is ordinary code. The compiler translates it and it runs.
- `//` marks where the comment begins.
- An end-of-line comment: `...` runs to the end of the line and is ignored.

There is no closing marker. The end of the line ends the comment, so the next line is code again.

WORKED EXAMPLE Labeling units with end-of-line comments

Each statement below has a note on its right. Look for where each note begins.

```
1 double price = 4.5; // price of one smoothie, in dollars
2 int count = 3; // smoothies ordered
3 double total = price * count; // 3 times 4.5
4 System.out.println(total);
```

13.5

1. Line 1: `double price = 4.5;` runs; the note "price of one smoothie, in dollars" tells the reader the unit.
2. Line 2: `int count = 3;` runs; the note says what is being counted.
3. Line 3: `double total = price * count;` runs. A `double` times an `int` gives a `double` (Topic 1.3), so `total` is 13.5. The note `3 times 4.5` is ignored, even though it looks like arithmetic.
4. Line 4 prints 13.5.

Now vary the order on a line. In `int pages = 320; // number of pages`, the code comes first and runs. Flip it, and the code is gone:

WATCH OUT

Putting code after `//` on the same line. A student types the note first and the declaration after it:

CommentSwallowsCode.java

```
1 public class CommentSwallowsCode
2 {
3     public static void main(String[] args)
4     {
5         // number of pages in the book int pages = 320;
6         System.out.println(pages);
7     }
8 }
```

Everything after `//` on line 5 is a comment, including `int pages = 320;`. So `pages` is never declared, and line 6 cannot use it. The compiler reports:

```
CommentSwallowsCode.java:6: error: cannot find symbol
    System.out.println(pages);
                        ^
symbol:   variable pages
```

(The exact wording can vary between Java versions.) The fix is to put the code first and the note after it:

```
1 int pages = 320; // number of pages in the book
2 System.out.println(pages);
```

This prints 320. A `//` comment is always the *last* thing on its line.

PAUSE AND TRY

Question 3 Multiple choice

In the line `double tip = bill * 0.2; // 20% of bill * 0.2`, which part does the compiler translate?

- A. Nothing; a line that contains `//` is skipped completely.
- B. `double tip = bill * 0.2;` only
- C. The whole line, including `20% of bill * 0.2`

D. Only 20% of bill * 0.2, because it comes last

Question 4 Debugging

This code is supposed to start an album with 12 stamps, add 6, and print 18. It compiles, but it prints 12. Find the error and correct it.

```
1 int stamps = 12; // stamps in the album stamps += 6;
2 System.out.println(stamps);
```

1.8.3 Block Comments

Sometimes a note needs several lines: a description of what a whole program does, or a few sentences about how a tricky step works. You could start every line with `//`, but there is a tool made for longer notes.

A **block comment** starts with `/*` and ends with `*/`. Everything between the two markers is a comment, no matter how many lines it covers. Unlike `//`, the end of a line does **not** end a block comment; only `*/` does.

WORKED EXAMPLE Describing a program with a block comment

Lines 1 through 5 are one block comment. Look for where it opens and closes.

GardenOrder.java

```
1 /*
2  * GardenOrder
3  * Works out how many seed packets to buy for a garden.
4  * Each row of the garden needs 4 packets.
5  */
6 public class GardenOrder
7 {
8     public static void main(String[] args)
9     {
10         int rows = 6;
11         int packets = rows * 4;
12         System.out.println(packets);
13     }
14 }
```

24

1. Line 1 opens the comment with `/*`. From here on, the compiler ignores everything.

- Lines 2 to 4 are the note. The `*` at the start of each line is only decoration that many programmers use to make the block easy to see. It is ignored like the rest.
- Line 5 closes the comment with `*/`. Line 6 is code again.
- The program runs: `rows` is 6, `packets` is $6 \times 4 = 24$, and line 12 prints 24.

Now use a block comment for a different job: switching off several lines at once.

```
1 int rows = 6;
2 /*
3 rows += 2;
4 System.out.println("Added two rows");
5 */
6 int packets = rows * 4;
7 System.out.println(packets);
```

24

Lines 3 and 4 are valid statements, but they sit between `/*` on line 2 and `*/` on line 5. Neither runs: `rows` stays 6 (not 8), `Added two rows` is never printed, and the output is still 24. Commenting out four lines with `//` would take four markers; a block comment takes one pair.

WATCH OUT

Forgetting the closing `*/`. Since only `*/` ends a block comment, leaving it out turns the rest of the file into a comment:

UnclosedComment.java

```
1 public class UnclosedComment
2 {
3     /*
4         * Prints a reminder before the test.
5
6     public static void main(String[] args)
7     {
8         System.out.println("Bring a calculator!");
9     }
10 }
```

The comment opened on line 3 never closes, so the compiler swallows `main`, the `println`, and both closing braces. It reports:

```
UnclosedComment.java:3: error: unclosed comment
    /*
    ^
```

usually followed by a second error saying the end of the file was reached too early (wording can vary between Java versions). Fix it by adding `*/` on its own line after the note, before `public static void main`. Whenever you type `/*`, type `*/` right away and then fill in the note between them.

Here are the two kinds of comments so far, side by side:

Kind	Starts with	Ends at	Typical use
Single-line comment	//	The end of that line	Short note on one line, or after code
Block comment	/*	The next */	Notes several lines long; switching off a block

PAUSE AND TRY

Question 5 Tracing

What does this code print? Watch where the block comment ends.

```
1 int coins = 10;
2 /* coins += 5;
3 coins *= 2; */ coins -= 3;
4 System.out.println(coins);
```

Question 6 Multiple choice

A program has `/*` on line 4 and no `*/` anywhere after it. What happens?

- A. Only line 4 is a comment; the rest of the program compiles normally.
- B. The program compiles, but nothing after line 4 runs.
- C. The program does not compile, because the comment never ends.
- D. Java closes the comment automatically at the end of line 4.

1.8.4 Javadoc Comments and API Documentation

In Topic 1.7 you learned that a library's documentation tells you what its classes can do: the methods of `String` and `Math`, what each one needs, and what each one gives back. Someone had to write those pages. Writing them separately from the code would be risky, since the pages and the code could drift apart. So the people who write a class put the documentation **inside the source code**, right above each class and method, and a tool builds the pages from it.

A **Javadoc comment** starts with `/**` (a slash and **two** stars) and ends with `*/`. It is written directly above a class or a method and describes it. To the compiler, a Javadoc comment is just another comment and is ignored. But a separate tool that comes with Java, called **javadoc**, reads every `/** */` comment in a class and turns them into the documentation pages, known as the **API documentation**, that programmers read. The `String` documentation you looked at in Topic 1.7 was built this way from comments in the `String` class's source code.

KEY IDEA

A Javadoc comment, written `/** */` directly above a class or method, is ignored by the compiler like any comment, and is used by the javadoc tool to create API documentation.

WORKED EXAMPLE Inside the String class

Here is a **simplified** sketch of how the `String` class's source code begins. The real file is far longer, and its comments use different wording; this keeps only the shape. The bodies are left out, so this is for reading, not running.

```
1 /**
2  * A String holds a sequence of characters, such as "Hopper".
3  */
4 public class String
5 {
6     /**
7      * Returns the number of characters in this string.
8      *
9      * @return the number of characters in this string
10     */
11     public int length()
12     {
13         // ...the statements that count the characters...
14     }
15 }
```

Read it from the top:

1. The first comment, opened by `/**`, sits directly above `public class String`. It describes the whole class. The javadoc tool puts this sentence at the top of the `String` documentation page.
2. The second Javadoc comment sits directly above the method header `public int length()`. Its first sentence becomes the method's description on the page. (`public` is a word you can read past for now; method headers are taught fully in Topic 1.9.)
3. The line beginning with `@return` holds a Javadoc **tag**. A tag is a word starting with `@` that the javadoc tool treats specially. `@return` describes the value the method gives back, and the tool lists it under a "Returns" heading.
4. Inside the method's braces is an ordinary `//` comment. It is a note for whoever maintains `String`, not for users of the class, so the tool does not put it on the documentation page.

A second common tag is `@param`, which describes a value the method needs to be given. Here is a simplified Javadoc comment for a `Math` method you will use in Topic 1.11:

```
1 /**
2  * Returns the absolute value of a: the value without its minus sign.
3  *
4  * @param a    the number to use
5  * @return     the absolute value of a
6  */
7 public static int abs(int a)
```

The `@param` line names `a`, the value that goes in, and says what it is. The `@return` line says what comes out. (`public static` can be read past for now; those words are explained in later topics.) Javadoc comments work in your own programs too. This program has one above the class and one above `main`:

TipCalculator.java

```

1  /**
2   * Works out the tip on a restaurant bill.
3   */
4  public class TipCalculator
5  {
6      /**
7       * Prints a 20 percent tip on a 45-dollar bill.
8       */
9      public static void main(String[] args)
10     {
11         double bill = 45.0;
12         double tip = bill * 0.2;
13         System.out.println(tip);
14     }
15 }
```

9.0

The compiler skips lines 1 to 3 and 6 to 8, just as it skips any comment. The program stores 45.0 in `bill`, computes $45.0 \times 0.2 = 9.0$, and prints 9.0. The Javadoc comments change nothing about how the program runs; they only give the javadoc tool something to turn into documentation.

WATCH OUT

Writing `/*` when you mean `/**`. A comment that starts with `/*` and one that starts with `/**` look almost the same, and both compile, because the compiler ignores both. The difference shows up only in the documentation: the javadoc tool collects comments that start with `/**` and skips ordinary block comments. A student who writes a careful description in `/* ... */` above a method will find it missing from the documentation pages. Use `/**\*\*/` for documentation that users of a class should see, and `/* ... */` for notes about how the code works inside.

All three kinds of comments together:

Kind	Written as	Ignored by the compiler?	Used for
Single-line	<code>// ...</code>	Yes	A short note to the end of the line
Block	<code>/* ... */</code>	Yes	A note spanning several lines
Javadoc	<code>/** ... */</code>	Yes	Describing a class or method; builds API documentation

PAUSE AND TRY**Question 7** Multiple choice

A programmer wants the description of a method to appear in the API documentation for her class. Which comment should she write directly above the method header?

- A. `// Returns the total price in cents.`
- B. `/* Returns the total price in cents. */`
- C. `/** Returns the total price in cents. */`
- D. Any of these; the javadoc tool reads every comment.

Question 8 Short answer

Here is a simplified Javadoc comment from the `Math` class. Using only the comment, say what you must give this method, and what it gives back.

```
1 /**
2  * Returns the larger of two values.
3  *
4  * @param a   one number
5  * @param b   another number
6  * @return    the larger of a and b
7  */
8 public static int max(int a, int b)
```

1.8.5 Preconditions

A recipe for bread might say, "Start with the oven heated to 200 °C." The recipe does not tell you to check the oven; it assumes you already did. If you skip that, the recipe still runs its steps, and you get a pale, raw loaf. Code works the same way: much of it only works if something is already true before it starts.

You have already met a piece of code like that. In Topic 1.5 you rounded a `double` to the nearest whole number with `(int) (x + 0.5)`, and that trick works only when `x` is not negative. That "only when" is the idea of this subsection.

A **precondition** is a condition that has to be true right before a method or a segment of code runs, so that it does what its documentation says. Preconditions are usually stated in a comment. Here is the most important rule about them: **the method or code is not expected to check its own precondition**. Making sure the precondition is true is the job of whoever uses the code. If the precondition is false, all bets are off: the code may give a wrong answer, or the program may crash.

KEY IDEA

A precondition is what must be true before a method or code segment runs. The code does not check it for you; you are responsible for making it true before you use the code.

WORKED EXAMPLE A precondition that is met, and one that is broken

The comment on line 1 states the precondition for the rounding code.

```
1 // Precondition: temperature is not negative.
2 double temperature = 2.7;
3 int rounded = (int) (temperature + 0.5);
4 System.out.println(rounded);
```

3

With `temperature` 2.7, the precondition holds (2.7 is not negative). Line 3 adds 0.5 to get 3.2, and the `(int)` cast drops the decimal part, giving 3. The nearest whole number to 2.7 is 3, so the code worked.

Now change only the value, so the precondition is broken:

```
1 // Precondition: temperature is not negative.
2 double temperature = -2.7;
3 int rounded = (int) (temperature + 0.5);
4 System.out.println(rounded);
```

-2

Step	Value
<code>temperature</code>	-2.7
<code>temperature + 0.5</code>	-2.2
<code>(int)</code> drops the decimal	-2
Correct nearest whole number	-3

The code did not stop, warn, or check anything. It ran every step and printed -2, which is wrong: -2.7 is closer to -3. Casting drops the decimal part, which moves a positive number down but a negative number up, so the + 0.5 trick only works on the nonnegative side. The comment warned about exactly this, and it was the user's job to obey it.

Breaking a precondition does not always give a quiet wrong answer. Sometimes the program crashes. This segment finds a player's average points per game:

```
1 // Precondition: games is greater than 0.
2 int totalPoints = 84;
3 int games = 6;
4 System.out.println("Computing average...");
5 int average = totalPoints / games;
6 System.out.println(average);
```

```
Computing average...  
14
```

84 / 6 is 14, so all is well. Now suppose the player has not played yet, so `games` is 0:

```
1 // Precondition: games is greater than 0.  
2 int totalPoints = 84;  
3 int games = 0;  
4 System.out.println("Computing average...");  
5 int average = totalPoints / games;  
6 System.out.println(average);
```

```
Computing average...  
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

The first `println` runs, then line 5 asks Java to divide by zero. As you saw in Topic 1.1, that throws an exception and stops the program, so the average is never printed. (The full message also names the line, and its wording can vary between Java versions.) Again, the code did nothing to protect itself; the precondition comment was the only warning.

Methods in Java's libraries have preconditions too, written in their Javadoc comments. Here is a simplified version of the documentation for the `Math` class's square-root method, which you will use in Topics 1.10 and 1.11:

```
1 /**  
2  * Returns the square root of a.  
3  * Precondition: a is not negative.  
4  */  
5 public static double sqrt(double a)
```

No real number times itself is negative, so a negative number has no square root to give back. The method does not promise to check its precondition for you; if you pass a negative number, you do not get a usable answer. So before you call it, you make sure the argument is not negative. Here it is used correctly. (`Math.sqrt(area)` calls the method on the value of `area`; Topic 1.10 explains calls like this fully.)

```
1 double area = 49.0;  
2 double side = Math.sqrt(area);  
3 System.out.println(side);
```

```
7.0
```

`area` is 49.0, which is not negative, so the precondition holds and `sqrt` gives back 7.0.

WATCH OUT

"If the input were bad, the code would tell me." Students often assume code checks its own inputs and complains about bad ones. A precondition means the opposite: the code assumes the condition is true and goes ahead. The rounding example printed `-2` with no warning at all.

Before you run code that has a precondition, look at the values you are about to give it and confirm they meet the precondition yourself.

PAUSE AND TRY

Question 9 Multiple choice

The documentation for a method says, "Precondition: count is greater than 0." A program calls the method with `count` equal to 0. Which statement is true?

- A. The method will notice that `count` is 0 and print a warning.
- B. The program will not compile, because the precondition is false.
- C. The method may give a wrong result or crash; the caller was responsible for making `count` greater than 0.
- D. The method will change `count` to 1 so that its precondition is true.

Question 10 Tracing

What does this code print? Is that the nearest whole number to `temperature`?

```
1 // Precondition: temperature is not negative.
2 double temperature = -4.6;
3 int rounded = (int) (temperature + 0.5);
4 System.out.println(rounded);
```

1.8.6 Postconditions

A precondition tells you what you must do **before** using some code. The other half of the deal is what the code promises **after**. When you use `Math.sqrt` correctly, what exactly do you get? If a method changes something, what will be different when it finishes? Without that promise, you would have to read the method's code to find out, which is what documentation is meant to save you from.

A **postcondition** is a condition that is always true after a method or code segment finishes running (as long as its precondition was met). A postcondition describes the result: what value the method gives back, or what the current values of an object's attributes are afterward. Think of it as the method's promise. The precondition is your side of the deal; the postcondition is the method's side.

KEY IDEA

A postcondition is the guaranteed outcome after a method or code segment runs: the value it gives back, or the state its object's attributes are left in.

WORKED EXAMPLE Checking a postcondition on real output

The comments on lines 1 to 3 state both conditions for this segment. Read the postcondition, then check the output against it.

```

1 // Precondition: totalMinutes is not negative.
2 // Postcondition: hours holds the whole hours in totalMinutes,
3 // and minutes holds the leftover minutes, from 0 to 59.
4 int totalMinutes = 135;
5 int hours = totalMinutes / 60;
6 int minutes = totalMinutes % 60;
7 System.out.print(hours);
8 System.out.print(" h ");
9 System.out.print(minutes);
10 System.out.println(" min");

```

2 h 15 min

Line	What happens	Value
4	totalMinutes is set	135
5	135 / 60, an int division	hours is 2
6	135 % 60, the remainder	minutes is 15
7–10	Prints the parts on one line	2 h 15 min

The precondition held (135 is not negative), and the postcondition is true afterward: 2 whole hours, and 15 leftover minutes, which is between 0 and 59.

Now vary the input to test the edge of the promise: use 59 minutes, which is less than one hour. Changing line 4 to `int totalMinutes = 59;` prints:

0 h 59 min

`59 / 60` is 0 and `59 % 60` is 59. The postcondition still holds: 0 whole hours, and 59 leftover minutes, which is in the 0 to 59 range. A good postcondition holds for every input that meets the precondition, not just typical ones.

Postconditions appear in Javadoc comments too. In the `String` sketch from 1.8.4, the line "Returns the number of characters in this string" is a postcondition stated in terms of what is **returned**: after `length` runs, you are guaranteed a count of the characters. The `sqrt` method's documentation promises that it returns the square root of `a`.

A postcondition can also describe an object's **attributes** (from Topic 1.7, the data a class keeps in variables). Here is the documentation for a method in an invented `BankAccount` class, whose attribute is the account's balance. The rest of the header is taught in Topic 1.9; for now, focus on the comment.

```

1 /**
2  * Adds money to this account.
3  * Precondition: amount is greater than 0.
4  * Postcondition: the balance has gone up by amount.
5  */
6 public void deposit(double amount)

```

This method does not give back a value. Its promise is about the attribute: when it finishes, the balance is larger by exactly `amount`.

Question	Precondition	Postcondition
When is it true?	Right before the code runs	Right after the code finishes
Whose job is it?	The caller's: make it true	The method's: it promises this
<code>sqrt</code> example	<code>a</code> is not negative	Returns the square root of <code>a</code>
<code>deposit</code> example	<code>amount</code> is greater than 0	The balance has gone up by <code>amount</code>

WATCH OUT

Mixing up the two. A student writes "Postcondition: amount must be greater than 0." But that is a requirement on the value given to the method before it runs, so it is a precondition. Ask two questions about any condition: *When* is it true, before or after? And *who* makes it true, the caller or the method? A requirement the caller must meet beforehand is a precondition. A result the method guarantees afterward is a postcondition.

PAUSE AND TRY

Question 11 Multiple choice

A method that converts a test score to a letter grade is documented with four statements. Which one is a postcondition?

- A. The score must be between 0 and 100.
- B. The score given to the method is an `int`.
- C. The method returns one of the letters A, B, C, D, or F.
- D. The caller must not pass a negative score.

Question 12 Tracing

The segment from 1.8.6 is run with a new value. What does it print, and is its postcondition true afterward?

```
1 // Precondition: totalMinutes is not negative.
2 // Postcondition: hours holds the whole hours in totalMinutes,
3 // and minutes holds the leftover minutes, from 0 to 59.
4 int totalMinutes = 200;
5 int hours = totalMinutes / 60;
6 int minutes = totalMinutes % 60;
7 System.out.print(hours);
8 System.out.print(" h ");
9 System.out.print(minutes);
10 System.out.println(" min");
```

Putting it together

This complete program uses all three kinds of comments and documents a precondition and a postcondition. Before reading the walkthrough, find each comment and decide which kind it is.

PizzaShare.java

```

1  /**
2   * Splits pizza slices evenly among friends.
3   */
4  public class PizzaShare
5  {
6      /**
7       * Prints each friend's share and the slices left over.
8       */
9      public static void main(String[] args)
10     {
11         /**
12          * Precondition: friends is greater than 0.
13          * Postcondition: each friend's share and the leftover
14          * slices have been printed, and leftover is less than friends.
15          */
16         int slices = 16;    // 2 pizzas, 8 slices each
17         int friends = 5;
18
19         int perFriend = slices / friends;    // whole slices only
20         int leftover = slices % friends;
21
22         System.out.print("Each friend gets ");
23         System.out.println(perFriend);
24         // System.out.println("Enjoy!");
25         System.out.print("Slices left over: ");
26         System.out.println(leftover);
27     }
28 }

```

```

Each friend gets 3
Slices left over: 1

```

The comments.

Lines	Kind	What it is for
1–3	Javadoc	Describes the class; would appear in documentation
6–8	Javadoc	Describes main
11–15	Block	States the precondition and postcondition for the code below it
16, 19	End-of-line	Explain where 16 comes from and that division keeps whole slices
24	Single-line	A <code>println</code> that has been commented out

The run. The compiler skips every comment, leaving the statements on lines 16, 17, 19, 20, 22, 23, 25, and 26.

Line	What happens	Variables or output
16	slices is set	slices = 16
17	friends is set	friends = 5
19	16 / 5, an int division	perFriend = 3
20	16 % 5, the remainder	leftover = 1

22–23	Label, then the value	Each friend gets 3
24	Commented out: nothing happens	
25–26	Label, then the value	Slices left over: 1

Checking the conditions. The precondition held, since 5 is greater than 0. So the postcondition must be true, and it is: both values were printed, and the leftover (1) is less than the number of friends (5). If there were 5 or more slices left over, everyone could have had one more, so the division would have been wrong.

Breaking the precondition. If line 17 were `int friends = 0;`, the precondition would be false. Nothing in the program checks for that. Line 19 would divide by zero, Java would throw an `ArithmeticException`, and the program would stop before printing anything. The precondition comment is what tells the next programmer to make sure `friends` is never 0.

Vocabulary

Term	Definition	Example
comment	Text in a program written for people to read; ignored by the compiler and never run	<code>// points out of 20</code>
commenting out	Turning code into a comment so it does not run, without deleting it	<code>// tickets += 1;</code>
single-line comment	A comment from <code>//</code> to the end of that line	<code>int count = 3; // smoothies</code>
block comment	A comment from <code>/*</code> to the next <code>*/</code> , over any number of lines	<code>/* Each row needs 4 packets. */</code>
Javadoc comment	A comment from <code>/**</code> to <code>*/</code> placed above a class or method; used to build API documentation	<code>/** Returns the number of characters. */</code>
javadoc	The tool that turns Javadoc comments into API documentation pages	Builds the <code>String</code> documentation
tag	A word starting with <code>@</code> in a Javadoc comment; <code>@param</code> describes an input, <code>@return</code> the result	<code>@return the larger of a and b</code>
precondition	What must be true right before a method or code segment runs; the caller must make it true, and the code does not check it	<code>a</code> is not negative, for <code>sqrt</code>
postcondition	What is always true right after a method or code segment runs: what it returns, or the state of its object's attributes	The balance has gone up by <code>amount</code>

Chapter summary

- Comments are notes written for the original programmer and for other programmers. The compiler ignores them and they never run, so they never change a program's output; code inside a comment (commented out) does not run either.
- `//` makes a comment that ends at the end of its line. `/* */` makes a block comment that can cover many lines and ends only at `*/`.
- `/** */` makes a Javadoc comment. Written above a class or method, it describes it, and the javadoc tool turns these comments into API documentation like the pages for `String` and `Math`. Tags such as `@param` and `@return` describe what goes in and what comes out.
- A precondition is what must be true before a method or code segment runs. The code does not check it; the user is responsible, and breaking it can cause a wrong result or a crash.
- A postcondition is what is guaranteed to be true afterward, described in terms of the value returned or the values of the object's attributes.

Review questions

Question 13 Multiple choice

Which statement is true about all three kinds of Java comments (`//`, `/* */`, and `/** */`)?

- A. They are all used by the javadoc tool to create API documentation.
- B. They are all ignored by the compiler and never executed.
- C. They all end at the end of the line on which they start.
- D. They are all printed when the program runs, so users can read them.

Question 14 Tracing

What is the exact output of this program?

ReadingLog.java

```
1  /**
2   * Keeps count of the books a student has read.
3   */
4  public class ReadingLog
5  {
6      public static void main(String[] args)
7      {
8          int books = 2; // read in September
9          /* books += 4; read in October
10             books += 1; */
11          books *= 3; // books -= 1;
12          // books += 10;
13          System.out.print("Books: ");
14          System.out.println(books);
15      }
16  }
```

Question 15 Debugging

This code should record 5 cups of water, then add 3 glasses of 2 cups each, and print 11. It prints 5. Find the error and correct it.

```
1 int water = 5; // cups of water so far
2 /* Each glass holds 2 cups.
3    The student drinks 3 more glasses.
4    water += 2 * 3; */
5 System.out.println(water);
```

Question 16 Short answer

A library includes this documentation for a method in an invented `Ticket` class:

```
1 /**
2  * Works out the total cost of an order of tickets.
3  * Precondition: quantity is greater than 0, and priceCents is not negative.
4  * Postcondition: returns quantity times priceCents.
5  */
6 public static int totalCost(int quantity, int priceCents)
```

(a) Who is responsible for making sure `quantity` is greater than 0? (b) If a program passes 0 for `quantity`, what does the documentation promise about the result? (c) Which kind of comment is this, and how can you tell?

Question 17 Tracing

The minutes segment from 1.8.6 is run with a value that breaks its precondition. What does it print, and which part of the postcondition is false afterward?

```
1 // Precondition: totalMinutes is not negative.
2 // Postcondition: hours holds the whole hours in totalMinutes,
3 // and minutes holds the leftover minutes, from 0 to 59.
4 int totalMinutes = -135;
5 int hours = totalMinutes / 60;
6 int minutes = totalMinutes % 60;
7 System.out.print(hours);
8 System.out.print(" h ");
9 System.out.print(minutes);
10 System.out.println(" min");
```

Question 18 Multiple choice

A programmer writes this segment to split cookies evenly into boxes:

```
1 int perBox = cookies / boxes;
```

Which comment is the best precondition to write above it?

- A. `// Precondition: perBox holds the number of cookies in each box.`
- B. `// Precondition: boxes is not 0.`
- C. `// Precondition: the code checks that boxes is not 0.`
- D. `// Precondition: perBox is greater than 0.`

Reflection

Check each statement. If the answer is "not yet," reread the linked subsection.

- ☐ I can explain who comments are for and why they never change what a program prints. Reread: 1.8.1 Comments Are Notes for People
- ☐ I can tell which part of a line is code and which part is a `//` comment. Reread: 1.8.2 Single-Line Comments
- ☐ I can find where a `/* */` block comment starts and ends, even in the middle of a line. Reread: 1.8.3 Block Comments
- ☐ I can explain how `/** */` comments become API documentation and read `@param` and `@return`. Reread: 1.8.4 Javadoc Comments and API Documentation
- ☐ I can state a precondition and explain that the caller, not the code, must make it true. Reread: 1.8.5 Preconditions
- ☐ I can state a postcondition and tell it apart from a precondition. Reread: 1.8.6 Postconditions